White Paper

Intel Software Solutions Group

Zhai Lei

Intel® XML Parsing Accelerator with Intel® Streaming SIMD Extensions 4 (Intel® SSE4)

Intel® SSE4 is a new set of Single Instruction Multiple Data (SIMD) instructions that will be introduced in the 45nm Next Generation Intel® Core™2 processor family and improves the performance and energy efficiency of a broad range of applications.

Intel® SSE4.2 includes four instructions for "string and text processing" which use XMM registers to process string or text elements of up to 128-bits (16 bytes or 8 words). Each instruction uses an immediate byte to support a rich set of programmable controls.

This white paper provides an overview of the string and text processing instructions, describes how Intel® XML parsing can benefit from the Intel SSE4.2 instructions, achieving great performance speedups using string and text processing instructions.

April 2008

Intel Corporation

Contents

Figures

Introduction

Intel® Streaming SIMD Extensions 4 (Intel® SSE4) is a new set of Single Instruction Multiple Data (SIMD) instructions designed to improve the performance of various applications, such as video encoders, image processing, 3D games, and string/text processing. Intel SSE4 builds upon the Intel® 64 and IA-32 instruction set, the most popular and broadly used computer architecture for developing 32-bit and 64-bit applications. Intel SSE4 will be introduced in the 45nm Next Generation Intel® Core™2 processor family.

Intel SSE4.2 introduces four instructions for "string and text processing", which can accelerate many string and text related applications, typically, the functionalities of finding characters in a set, checking characters in a range set, comparing 2 strings and finding substring from a string.

XML related applications such as Intel XML parsing become more and more common and popular in current information system, especially web service and SOA environment. In such environments, string and text processing is critical performance bottleneck, since many operations such as sub-string searching and regular expression checking need many complex algorithms.

This white paper will describe how Intel XML parsing can benefit from the Intel SSE4 instructions, achieving great performance speedups using string and text processing instructions.

String and Text Processing Instructions in Intel® SSE4

Overview

String and text processing instructions in INTEL SSE4.2 allocates four instructions to provide a rich set of string and text processing capabilities that traditionally required many more instructions. These four instructions use XMM registers to process string or text elements of up to 128-bits (16 bytes or 8 words). Each instruction uses an immediate byte to support a rich set of programmable controls. A string-processing INTEL SSE4.2 instruction returns the result of processing each pair of string elements using either an index or a mask.

The capabilities of the string/text processing instructions include:

- Handling string/text fragments consisting of bytes or words, either signed or unsigned.
- Support for partial string or fragments less than 16 bytes in length, using either explicit length or implicit null-termination.
- Four types of string compare operations on word/byte elements.
- Up to 256 compare operations performed in a single instruction on all string/text element pairs.
- Built-in aggregation of intermediate results from comparisons.
- Programmable control of processing on intermediate results.

- Programmable control of output formats in terms of an index or mask.
- Bi-directional support for the index format.
- Support for two mask formats: bit or natural element width.
- Not requiring 16-byte alignment for memory operand.

The four INTEL SSE4.2 instructions that process text/string fragments are:

- PCMPESTRI — Packed compare explicit-length strings, return index in ECX/RCX.
- PCMPESTRM — Packed compare explicit-length strings, return mask in XMM0.
- PCMPISTRI — Packed compare implicit-length strings, return index in ECX/RCX.
- PCMPISTRM — Packed compare implicit-length strings, return mask in XMM0.

All four require the use of an immediate byte to control operation. The two source operands can be XMM registers or a combination of XMM register and memory address. The immediate byte provides programmable control with the following attributes:

- Input data format.
- Compare operation mode.
- Intermediate result processing.
- Output selection.

Depending on the output format associated with the instruction, the text/string processing instructions implicitly uses either a general-purpose register (ECX/RCX) or an XMM register (XMM0) to return the final result.

Two of the four text-string processing instructions specify string length explicitly. They use two general-purpose registers (EDX, EAX) to specify the number of valid data elements (either word or byte) in the source operands. The other two instructions specify valid string elements using null termination. A data element is considered valid only if it has a lower index than the least significant null data element.

Programming models

There are four common programming models for using string and text processing in an application. These models differ in how the string comparison operation is. Below is a summary of each programming model, followed by an example and implementation guidelines for the preferred programming model, ranges.

1. Equal any

   In this model, the application would find characters from a set. Operand 1 is defined less than 16 bytes or eight words as the target set, while, operand 2 is the input bytes or words stream for checking. Each unit in operand 2 is checked whether equal with any unit in operand 1, then, the result is calculated according to the control byte.

2. Ranges

   In this model, the application would find characters from ranges. Operand 1 is defined less than 16 bytes or 8 words as the target range set, while, operand 2 is the input bytes or words stream for checking. Each unit in operand 2 is checked whether is located in any range defined in operand 1, then, the result is calculated according to the control byte.

3. Equal each

   In this model, the application would compare whether two strings are equal. Operand 1 is defined less than 16 bytes or eight words as one input string, while, operand 2 is defined as another input string. Each unit in operand 2 is checked whether is equal with the same location unit in operand 1, then, the result is calculated according to the control byte.

4. Equal ordered

   In this model, the application would search the substring from input string. Operand 1 is defined less than 16 bytes or eight words as the target substring, while, operand 2 is the input bytes or words stream for checking. Each sequence in operand 2 is checked whether is equal with the substring in operand 1, then, the result is calculated according to the control byte.

Character Range Checking using String and Text Processing Instructions

XML documents are made up of storage units called entities, like Character Data, Element, Comment, CDATA Section, etc. Each type of entity has its own well-formed definition that is a serious of character range rules.[1] The main work of Intel XML parsing is to recognize these entities and their logic structures.

From Intel XML Parsing Accelerator, we found that character checking loop occupies more than 60% CPU cycles of the whole parsing process, depending on the property of benchmark. There are two kinds of important behavior in this loop, read bytes and check whether it is legal for its corresponding enity type. Without any parallel instructions for string comparison, this process must be implemented in serializing mode.

STTNI provides a parallel solution for byte or word comparison. It matches the optimization of bottleneck in XML parser. Therefore, the basic idea is implementing the parallel character check using Range model. While the enity rules are generally very complex which are defined inside Unicode standard, we cannot apply all of these range rules with STTNI, considering the overhead of the STTNI initialization and execution. The best situation is the one byte encoding range (like ASCII), so one STTNI instruction can check all the character in XMM register, and the worst should be characters in the register compose an encoding sequence, each item in which has its own different range rule. In addition, STTNI instructions can define up to 16 bytes characters for parallel check, so we had better provide characters as many as possible to extremely utilize the STTNI power.

So the principal of accelerate Intel XML parsing using STTNI could be concluded as following:

Principle 1: The character to be checked should be the longer the better.

Principle 2: The character rule logic should be simple and cover most of the popular XML documents.

According to Principle 1, we choose the CharData, Comments, CDATA Sections, Element/ Attribute Name, and Attribute Value to apply the STTNI optimization, since normally their content length is not very short. According to Principle2, we simplify the

[1] For more details on XML documents, see the W3C XML spec Extensible Markup Language (XML) 1.0 (Fourth Edition) and Namespaces in XML 1.0 (Second Edition).

entity rules according to the concrete type of rule characteristic, to ensure using the least STTNI instructions to check the most part of the rule.

Figure 1 shows an example of Character Data rules simplification.[1] We selected full 1 and 2 bytes encoding part, major part of 3 bytes encoding part, and eliminate the complex and infrequently used 4 bytes encoding part. In the simplified 2 and 3 bytes encoding parts, we should well utilize the common range [\x80-\xBE], only when it fails we need to check the range [\x80-\xBF], which can extremely save the STTNI execution times. The detail is shown by the code in Figure 3.

Figure 1. Sample Character Data Rules Simplification

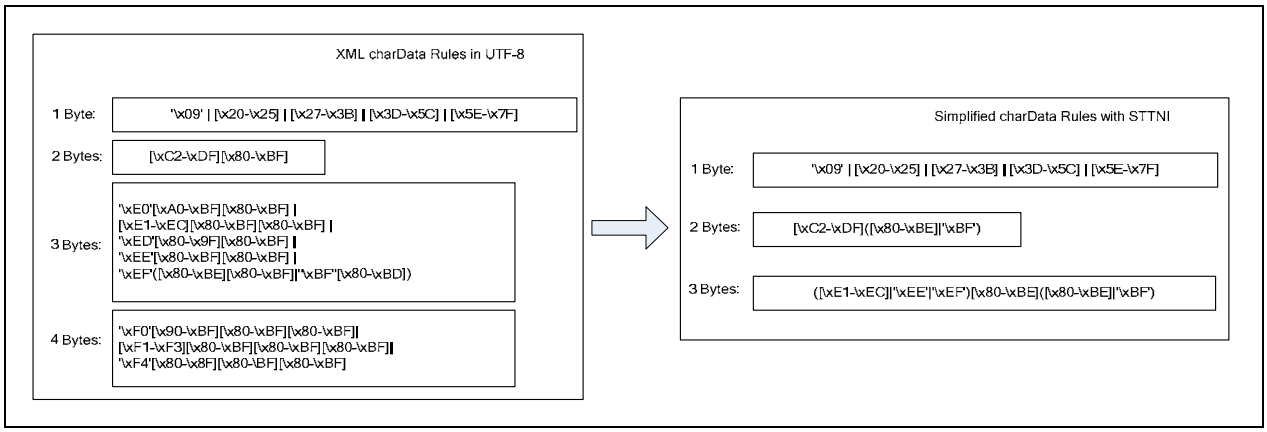

Figure 2 shows the sample implementation of the Character Data parsing acceleration using PCMPESTRI, PCMPESTRM instructions with Range/Equal Any Models. Firstly, we locate the Character Data part in the XMM register with the nonCharaData characters. Then we check whether they all hit the 1 byte encoding range (ASCII range), if so continue to process the next data, otherwise invoke *recogUnicodeRange* (Figure 3) function to check the non hint part with the 2 and 3 bytes encoding short-cut, if it still doesn't hit, fall back to the normal checking logic. When checking the rules that more than 2 bytes, we should first ensure that characters in XMM register doesn't contain the incomplete encoding character, as shown in Figure 4.

Figure 2. Sample Implementation of Character Data Rules Check with STTNI

```
1.  #include "nmmintrin.h"
2.  #include "emmintrin.h"
3.
4.  #define STTNISTRLENLIMIT 4
5.
6.  void scanCharDataContentwithSTTNI(SAX2Processor* saxProcessor) {
7.      unsigned int length = yylim - yycur;
8.      unsigned char* data = (unsigned char*)yycur;
9.
10.     if( *data == '<' || *data == '&' || *data == ']') return;
11.
12.     unsigned int dataLen = 0;
13.     // initialize the one byte encoding rule and nonCharaData rule
14.     const                    __m128i                    asciiCharData                =
    _mm_set_epi8(0,0,0,0,0,0,'\x7F','\x5E','\x5C','\x3D',
    '\x3B','\x27','\x25','\x20','\t','\t');
```

1 0x0A', '\0xOD' have special new Line processing, so we don't put them in the common Character Data rules.

```
15.     const __m128i nonCharData = _mm_set_epi8(0,0,0,0,0,0,0,0,0,0,0,'\x5D','\x3C',
    '\x26','\x0D','\x0A');
16.
17.     do {
18.         // special new line processing for '\x0A','\x0D'
19.         if( *data == '\n' ) {
20.             saxProcessor->newLine((char*)data);
21.             data++; length--;
22.         } else if(*data == '\r') {
23.             saxProcessor->newLine((char*)data);
24.             if( *(data+1) == '\n' ) {
25.                 data += 2; length -= 2; yycur++;
26.             } else {
27.                 *data = '\n'; data++; length--;
28.             }
29.         }
30.
31.         while( length > 0 ) {
32.             if( length >= 16 ) dataLen = 16;
33.             else dataLen = length;
34.
35.             const __m128i mData = _mm_loadu_si128((__m128i*)data);
36.             // locate the Character Data part with the nonCharaData characters
37.             int index = _mm_cmpestri(nonCharData, 5, mData, dataLen,
    SIDD_CMP_EQUAL_ANY);
38.             if( index == 0 ) break;
39.             if( index > dataLen ) index = dataLen;
40.             bool shouldBreak = index < dataLen ? true : false;
41.             // check the one byte encoding rule(ASCII)
42.             unsigned int mask = _mm_cvtsi128_si32(_mm_cmpestrm(asciiCharData, 10,
    mData, index, SIDD_CMP_RANGES|SIDD_MASKED_NEGATIVE_POLARITY));
43.             // if not all hit ASCII, continue to check other Unicode rules
44.             if( mask == 0 || recogUnicodeRange(mData, index, ~mask)) {
45.                 data += index;
46.                 length -= index;
47.                 if( shouldBreak ) break;
48.             } else {
49.                 break;
50.             }
51.         }
52.
53.         unsigned int passLen = (char*)data - yycur;
54.         if( passLen == 0 ) break;
55.         // report Character Data to user
56.         saxProcessor->reportCharDataContent(yycur, passLen);
57.         yycur += passLen;
58.         YYSWITCHBUFFER;
59.     } while( length >= STTNISTRLENLIMIT && (*data == '\n' || *data == '\r') );
60. }
```

Figure 3. Character Data Rules Check in UTF-8 Except ASCII Range

```
1.  bool recogUnicodeRange(const __m128i data, int& dataLength, unsigned int mask) {
2.
3.      //first check whether in the 2 bytes encoding range
4.
5.      const __m128i Unicode_80_BE = _mm_set_epi8(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, '\xBE', '\x80');
6.      unsigned int mask_80_BE = _mm_cvtsi128_si32(_mm_cmpestrm(Unicode_80_BE, 2, data,
    dataLength, SIDD_CMP_RANGES));
7.
8.      const __m128i Unicode_C2_DF = _mm_set_epi8(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, '\xDF', '\xC2');
9.      unsigned int mask_C2_DF = _mm_cvtsi128_si32(_mm_cmpestrm(Unicode_C2_DF, 2, data,
    dataLength, SIDD_CMP_RANGES));
10.
11.     if( mask_C2_DF > 0 ) {
12.         checkIncompleteBytes(mask_C2_DF, mask, dataLength, 1);
13.
14.         if( mask_C2_DF > 0 ) {
15.             unsigned int mask_C2_DF_2 = mask_C2_DF << 1;
16.             if( (mask_C2_DF_2 & mask_80_BE) != mask_C2_DF_2 ) {
17.                 const __m128i Unicode_80_BF = _mm_set_epi8(0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, '\xBF', '\x80');
18.                 unsigned                int                mask_80_BF                =
    _mm_cvtsi128_si32(_mm_cmpestrm(Unicode_80_BF, 2, data, dataLength, SIDD_CMP_RANGES));
19.                 if( (mask_C2_DF_2 & mask_80_BF) != mask_C2_DF_2 ) {
20.                     return false;
21.                 }
22.             }
23.             mask |= mask_C2_DF;
24.             mask |= mask_C2_DF_2;
25.             if( mask == 0xFFFFFFFF ) {
26.                 return true;
27.             }
28.         } else {
29.             if( dataLength <= 0 ) return false;
30.             if( mask == 0xFFFFFFFF ) return true;
31.         }
32.     }
33.
34.     //then check whether in the 3 bytes encoding range
35.
36.     const __m128i Unicode_E1_EC_EE_EF = _mm_set_epi8(0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    '\xEF', '\xEF', '\xEE', '\xEE', '\xEC', '\xE1');
37.     unsigned                int                mask_E1_EC_EE_EF                =
    _mm_cvtsi128_si32(_mm_cmpestrm(Unicode_E1_EC_EE_EF,      6,      data,      dataLength,
    SIDD_CMP_RANGES));
38.
39.     if( mask_E1_EC_EE_EF > 0 ) {
40.         checkIncompleteBytes(mask_E1_EC_EE_EF, mask, dataLength, 2);
41.
42.         if( mask_E1_EC_EE_EF > 0 ) {
43.             unsigned int mask_E1_EC_EE_EF_2 = mask_E1_EC_EE_EF << 1;
44.             unsigned int mask_E1_EC_EE_EF_3 = mask_E1_EC_EE_EF << 2;
```

```
45.
46.             if( (mask_E1_EC_EE_EF_2 & mask_80_BE) == mask_E1_EC_EE_EF_2 ) {
47.                 if( (mask_E1_EC_EE_EF_3 & mask_80_BE) != mask_E1_EC_EE_EF_3 ) {
48.                     const __m128i Unicode_80_BF = _mm_set_epi8(0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, '\xBF', '\x80');
49.                     unsigned              int               mask_80_BF               =
    _mm_cvtsi128_si32(_mm_cmpestrm(Unicode_80_BF, 2, data, dataLength, SIDD_CMP_RANGES));
50.                     if( (mask_E1_EC_EE_EF_3 & mask_80_BF) != mask_E1_EC_EE_EF_3 ) {
51.                         return false;
52.                     }
53.                 }
54.                 mask |= mask_E1_EC_EE_EF;
55.                 mask |= mask_E1_EC_EE_EF_2;
56.                 mask |= mask_E1_EC_EE_EF_3;
57.                 if( mask == 0xFFFFFFFF ) {
58.                     return true;
59.                 }
60.             } else {
61.                  return false;
62.             }
63.         } else {
64.             if( dataLength <= 0 ) return false;
65.             if( mask == 0xFFFFFFFF ) return true;
66.         }
67.     }
68.
69.     return false;
70. }
```

Figure 4. Incomplete Encoding Bytes Check in XMM Register

```
1.  inline void checkIncompleteBytes(unsigned int& mask, unsigned int& tmask, int& length,
    unsigned int num) {
2.      for(unsigned int i = num; i > 0; i--) {
3.          if( (mask & (1<<(length-i))) > 0 ) {
4.              for(unsigned int j = i; j > 0; j--) {
5.                  mask &= ~(1<<length-j);
6.                  tmask |= (1<<length-j);
7.              }
8.              length -= i;
9.              break;
10.         }
11.     }
12. }
```

Performance Measurement

We use the Intel XML parsing benchmark to measure the performance of string and text processing for applications. The optimized character range checking implementation described in the previous section is used. .

Intel XML parsing benchmark is composed of 31 XML files. Some of them are public available for benchmarking , some of them are typical customer cases. The file size varies from 0.5K to 5.9M. Most of them are composed of the characters in ASCII range.

After applying the STTNI optimization, Intel XML parser gets overall 25% performance improvement. For some cases, the performance gains can be achieved up to 70%. The longer the entity content which hit the STTNI shortcut is and the less STTNI execution times are, the better the performance will be.

Conclusion

The use of String and Text Processing Instructions in Intel SSE4 was shown to improve the performance of the character range checking in Intel XML parsing. For files ranging from 0.5k to 5.9M the STTNI optimization yielded an overall performance improvement of 1.25X with some optimizations getting a 1.7X improvement in performance.

About the Author

Zhai Lei is a Senior Software Engineer in the Software Solutions Group at Intel Corporation working for the XML Engineering team. His current focus is on Intel XML parsing acceleration, on both algorithm level and instruction level, such as Intel SSE optimizations for XML processing. His email is lei.zhai@intel.com.